DCI_Closed: a Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets

Claudio Lucchese
ISTI "A. Faedo"
Consiglio Nazionale delle Ricerche (CNR)
Pisa, Italy.
email: claudio.lucchese@isti.cnr.it

Salvatore Orlando
Computer Science Dept.
Università Ca' Foscari
Venezia, Italy.
email: orlando@dsi.unive.it

Raffaele Perego
ISTI "A. Faedo"
Consiglio Nazionale delle Ricerche (CNR)
Pisa, Italy.
email: raffaere.perego@isti.cnr.it

Abstract

One of the main problems raising up in the frequent closed itemsets mining problem is the duplicate detection. In this paper we propose a general technique for promptly detecting and discarding duplicate closed itemsets, without the need of keeping in the main memory the whole set of closed patterns.

Our approach can be exploited with substantial performance benefits by any algorithm that adopts a vertical representation of the dataset. We implemented our technique within a new depth-first closed itemsets mining algorithm. The experimental evaluation demonstrates that our algorithm outperforms other state of the art algorithms like CLOSET+ and FPCLOSE.

1. Introduction

Frequent itemsets mining is the most important and demanding task in many data mining applications. To describe the mining problem we introduce the following notation. Let $\mathcal{I} = \{a_1, ..., a_M\}$ be a finite set of items, and $\mathcal{D}$ a finite set of transactions (the dataset), where each transaction $t \in \mathcal{D}$ is a list of *distinct* items $t = \{i_0, i_1, ..., i_T\}$, $i_j \in \mathcal{I}$. A k-itemset is a sequence of k *distinct* items $I = \{i_0, i_1, ..., i_k\} \mid i_j \in \mathcal{I}$, sorted on the basis of some total order relation between item literals. The number of transactions in the dataset including an itemset I is defined as the *support* of I (or *supp*(I)). Mining all the frequent itemsets from $\mathcal{D}$ requires to discover all the itemsets having support higher than (or equal to) a given threshold *min_supp*.

The paper is organized as follows. In Sect. 2 we introduce closed itemsets and describe a framework for mining them. This framework is shared by all the algorithms surveyed in Sect. 3. In Sect. 4 we formalize the problem of duplicates and propose our technique. Section 5 proposes an implementation of our technique and discusses the experimental results obtained. Follow some concluding remarks.

2. Closed itemsets

The concept of closed itemset is based on the two following functions f and g:

$$f(T) = \{i \in \mathcal{I} \mid \forall t \in T, i \in t\}$$
$$g(I) = \{t \in \mathcal{D} \mid \forall i \in I, i \in t\}$$

where T and I, $T \subseteq \mathcal{D}$ and $I \subseteq \mathcal{I}$ are, respectively, subsets of all the transactions and items occurring in dataset $\mathcal{D}$. Function f returns the set of itemsets included in all the transactions in T, while function g returns the set of transactions supporting a given itemset I. Since the set of transaction $g(I)$ can be represented by a list of *transaction identifiers*, we refer to $g(I)$ as the *tid-list* of I. We can introduce the following definition:

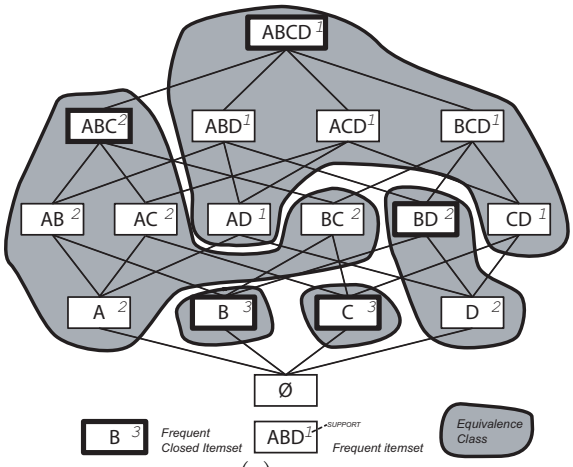
Definition 1 *An itemset I is said to be* closed *if and only if*

$$c(I) = f(g(I)) = f \circ g(I) = I$$

where the composite function $f \circ g$ is called Galois operator or closure operator.

The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belong to the same equivalence class *iff* they have the same closure, i.e. they are supported by the same set of transactions. We can also show that an itemset I is closed if no superset of I with the same support exists. Thus, a closed itemset is also the maximal itemset of an equivalence class. Mining all these maximal elements of each equivalence class corresponds to mine all the closed itemsets.

Figure 1.(a) shows the lattice of frequent itemsets derived from the simple dataset reported in Fig. 1.(b), mined with $min\_supp = 1$. We can see that the itemsets with the same closure are grouped in the same equivalence class. Each equivalence class contains elements sharing the same supporting transactions, and closed itemsets are their maximal elements. Note that the number of closed itemsets (five) is remarkably lower than the number of frequent itemsets (fifteen).



(a)

TID	items			
1	B	D		
2	A	B	C	D
3	A	B	C	
4	C			

(b)

Figure 1. (a) Lattice of frequent itemsets with closed itemsets and equivalence classes given by the dataset (b).

The algorithms for mining frequent closed itemsets adopt a strategy based on two main steps: *Search space browsing*, and *Closure calculation*. In fact, they *browse* the search space by traversing the lattice of frequent itemsets from an equivalence class to another, and they calculate the *closure* of the frequent itemsets visited in order to determine the maximal elements (closed itemsets) of the corresponding equivalence classes. Let us analyze in some depth these two phases.

Browsing the search space. The goal of an effective browsing strategy should be to devise a spanning tree over the lattice of frequent itemsets, visiting exactly a single itemset in each equivalence class. We could in fact mine all the closed itemsets by calculating the closure of just an itemset per equivalence class. Let us call the itemsets used to compute closures during the visit *closure generators*.

Some algorithms choose the minimal elements (or *key patterns*) of each equivalence class as closure generators. Key patterns form a lattice, and this lattice can be easily traversed with a simple apriori-like algorithm. Unfortunately, an equivalence class can have more than one minimal element leading to the same closed itemset. For example, the closed itemset $\{ABCD\}$ of Fig. 1 may be mined twice, since it can be obtained as closure of the two minimal elements of its equivalence class $\{AD\}$ and $\{CD\}$.

Other algorithms use instead a different technique that we call *closure climbing*. As soon as a generator is devised, its closure is computed, and new generators are built as supersets of the closed itemset discovered. Since closed itemsets are maximal elements, this strategy always guarantees to jump from an equivalence class to another. Unfortunately, it does not guarantee that the new generators belong to equivalence classes that were not previously visited.

Regardless of the strategy adopted, some kind of duplicate check has thus to be introduced. A naive approach to check for duplicates is to search for each generated closed itemset among all the ones mined so far. Indeed, in order to avoid to perform a lot of expensive closure operations, several algorithms exploit the following lemma:

Lemma 1 *Given two itemsets X and Y, if $X \subset Y$ and $supp(X) = supp(Y)$ (i.e. $|g(X)| = |g(Y)|$), then $c(X) = c(Y)$.*

Proof. If $X \subset Y$, then $g(Y) \subseteq g(X)$. Since $|g(Y)| = |g(X)|$ then $g(Y) = g(X)$. $g(X) = g(Y) \Rightarrow f(g(X)) = f(g(Y)) \Rightarrow c(X) = c(Y)$.
□

Therefore, given a generator X, if we find an already mined closed itemsets Y that set-includes X,

and the supports of Y and X are identical, we can conclude that $c(X) = c(Y)$. Hence we can prune the generator X without actually calculating its closure. Also this duplicates checking strategy is however expensive, both in time and space. In time because we may need to search for the inclusion of each generator in a huge number of closed itemsets, in space because to perform it we need to keep all the closed itemsets in the main memory. To reduce such costs, closed sets can be stored in compact prefix tree structures, indexed by one or more levels of hashing.

Calculating Closures. To calculate the closure of an itemset X, we have to apply the Galois operator c. Applying c requires to intersect all the transactions of the dataset including X. Another way to calculate the closure is given by the following lemma:

Lemma 2 *Given an itemset X and an item i, if $g(X) \subseteq g(i) \Rightarrow i \in c(X)$.*

Proof. Since $g(X \cup i) = g(X) \cap g(i)$, $g(X) \subseteq g(i) \Rightarrow g(X \cup i) = g(X)$. Therefore, if $g(X \cup i) = g(X)$ then $f(g(X \cup i)) = f(g(X)) \Rightarrow c(X \cup i) = c(X) \Rightarrow i \in c(X)$. □

From the above lemma, we know that if $g(X) \subseteq g(i)$, then $i \in c(X)$. Therefore, by performing this inclusion check for all the items in $\mathcal{I}$ not included in X, we can *incrementally* compute $c(X)$. Note that, since the set $g(i)$ can be represented by the *tid-list* associated with i, this suggests the adoption of a vertical format for the input dataset in order to efficiently implement the inclusion check: $g(X) \subseteq g(i)$.

The closure calculation can be performed off-line or on-line. In the first case we firstly retrieve the complete set of generators, and then we calculate their closures. In the second case, as soon as a new generator is discovered, its closure is computed on-the-fly.

The algorithms that compute closures on-line are generally more efficient. This is because they can adopt the *closure climbing* strategy, according to which new generators are created recursively from closed itemsets. These generators are likely longer than key patterns, which are the minimal itemsets of the equivalence class and thus are the shorter possible generators. Obviously, the longer the generator is, the fewer checks (on further items to add) are needed to get its closure.

3. Related Works

The first algorithm proposed for mining closed itemsets was A-CLOSE [5] (N. Pasquier, et al.). A-CLOSE first browses level-wise the frequent itemsets lattice by means of an Apriori-like strategy, and mines all the minimal elements of each equivalence class. Since a k-itemset is a key pattern if and only if no one of its $(k-1)$-subsets has the same support, minimal elements are discovered with an intensive subset checking. In its second step, A-CLOSE calculates the closure of all the minimal generators previously found. Since a single equivalence class may have more than one minimal itemsets, redundant closures may be computed. A-CLOSE performance suffers from the high cost of the off-line closure calculation and the huge number of subset searches.

The authors of FP-Growth [2] (J. Han, et al.) proposed CLOSET [6] and CLOSET+ [7]. These two algorithms inherit from FP-Growth the compact FP-Tree data structure and the exploration technique based on recursive conditional projections of the FP-Tree. Frequent single items are detected after a first scan of the dataset, and with another scan the pruned transactions are inserted in the FP-Tree stored in the main memory. With a depth first browsing of the FP-Tree and recursive conditional FP-Tree projections, CLOSET mines closed itemsets by closure climbing, and growing up frequent closed itemsets with items having the same support in the conditional dataset. Duplicates are discovered with subset checking by exploiting Lemma 2. Thus, all closed sets previously discovered are kept in the main memory, and are indexed by a two level hash. CLOSET+ is similar to CLOSET, but exploits an adaptive behaviour in order to fit both sparse and dense datasets. As regards the duplicate detection technique, CLOSET+ introduces a new one for sparse datasets named *upward checking*. This technique consists in the intersection of every path of the initial FP-Tree leading to a candidate closed itemset X, if such intersection is empty then X is actually closed. The rationale for using it only in sparse dataset is that the transactions are short, a thus the intersections can be performed quickly. Note that with dense dataset, where the transactions are usually longer, closed itemsets equivalence classes are large and the number of duplicates is high, such technique is not used because of its inefficiency, and CLOSET+ steps back using the same strategy of CLOSET, i.e. storing every mined closed itemset.

FPCLOSE [1], which is a variant of CLOSET+, resulted to be the best algorithm for closed itemsets mining presented at the ICDM 2003 Frequent Itemset Mining Implementations Workshop.

CHARM [9] (M. Zaki, et al.) performs a bottom-up depth-first browsing of a prefix tree of frequent itemsets built incrementally. As soon as a frequent itemset is generated, its tid-list is compared with those of the other itemsets having the same parent. If one tid-list includes another one, the associated nodes are merged

since both the itemsets surely belong to the same equivalence class. Itemset tid-lists are stored in each node of the tree by using the diff-set technique [8]. Since different paths can however lead to the same closed itemset, also in this case a duplicates pruning strategy is implemented. CHARM adopts a technique similar to that of CLOSET, by storing in the main memory the closed itemsets indexed by single level hash.

According to our classification, A-CLOSE exploits a key pattern browsing strategy and performs off-line closure calculations, while CHARM, CLOSET+ and FP-CLOSE are different implementations of the same closure climbing strategy with incremental closure computation.

4. Removing duplicate generators of closed itemsets

In this Section we discuss a particular visit of the lattice of frequent sets used by our algorithm to identify *unique generators* of each equivalence class, and compute all the closed patterns through the minimum number of closure calculations.

In our algorithm, we use *closure climbing* to browse the search space, find generators and compute their closure. As soon as a generator is found, its closure is computed, and new generators are built as supersets of the closed itemset discovered so far. So, each generator *gen* browsed by our algorithm can be generally represented as $gen = Y \cup i$, where Y is a closed itemset, and i, $i \notin Y$ is an item in $\mathcal{I}$[1].

Looking at Figure 1.(a), we can unfortunately discover multiple generators $gen = Y \cup i$, whose closures produce an identical closed itemset. For example, we have four generators, $\{A\}$, $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$, whose closure is equal to the closed itemsets $\{A, B, C\}$. Note that all these generators have the form $Y \cup i$, since they can be obtained by adding a single items to a smaller closed itemset, namely $\emptyset$, $\{B\}$ and $\{C\}$.

The technique exploited by our algorithm to detect duplicate generators exploits a *total lexicographic order* relation $\prec$ between all the itemsets of our search space[2]. Since there exist a relation $\prec$ between each pair of k-itemsets, in order to avoid duplicate closed itemsets, we do not compute the closure of the generators that do not result *order preserving* according to the definition below.

Definition 2 *A generator $X = Y \cup i$, where Y is a closed itemset and $i \notin Y$, is said to be* order preserving *one iff $i \prec (c(X) \setminus X)$.*

The following Theorem shows that, for any closed itemset Y, it is possible to find a sequence of order preserving generators in order to climb a sequence of closure itemsets and arrive at Y. The following Corollary states that this sequence is unique.

Theorem 1 *For each closed itemset $Y \neq c(\emptyset)$, there exists a sequence of n ($n \geq 1$) items $i_0 \prec i_1 \prec ... \prec i_{n-1}$ such that*

$$\{gen_0, gen_1, \ldots, gen_{n-1}\} = \{Y_0 \cup i_0, Y_1 \cup i_1, \ldots, Y_{n-1} \cup i_{n-1}\}$$

where the various gen_i are order preserving generators*, with $Y_0 = c(\emptyset)$, $Y_{j+1} = c(Y_j \cup i_j)$ $\forall j \in [0, n-1]$ and $Y = Y_n$.*

Proof. First of all, we show that given a generic generator $gen \subseteq Y$, $c(gen) \subseteq Y$. More formally, if $\exists Y'$ such that Y' is a closed itemset, and $Y' \subset Y$, and we extend Y' with an item $i \in Y \setminus Y'$ to obtain $gen = Y' \cup i \subseteq Y$, then $\forall j \in c(gen)$, $j \in Y$.

Note that $g(Y) \subseteq g(gen)$ because $gen \subseteq Y$. Moreover, if $j \in c(gen)$, then $g(c(gen)) \subseteq g(j)$. Thus, since $g(Y) \subseteq g(gen)$, then $g(Y) \subseteq g(j)$ also holds, so that $j \in c(Y)$ too. So, if $j \notin Y$ hold, Y would not be closed, and this is in contradiction with the hypothesis.

As regards the proof of the Theorem, we show it by constructing a sequence of closed itemsets and associated generators having the properties stated above.

We have that $Y_0 = c(\emptyset)$. All the items in Y_0 appear in every transaction of the dataset and therefore by definition of closure they must be included also in Y, i.e. $Y_0 \subseteq Y$.

Since $Y_0 \neq Y$ by definition, we choose $i_0 = \min_{\prec}(Y \setminus Y_0)$, i.e. i_0 is the smallest item in $\{Y \setminus Y_0\}$ with respect to the lexicographic order $\prec$, in order to create the first order preserving generator $\{Y_0 \cup i_0\}$. Afterwards we calculate $Y_1 = c(Y_0 \cup i_0) = c(gen_0)$.

Once Y_1 is found, if $Y_1 = Y$ we stop.

Otherwise we choose $i_1 = \min_{\prec}(Y \setminus Y_1)$, where $i_0 \prec i_1$ by construction, in order to build the next *order preserving* generator $gen_1 = Y_1 \cup i_1$ and we calculate $Y_2 = c(Y_1 \cup i_1) = c(gen_1)$.

Once Y_2 is found, if $Y_2 = Y$ we stop, otherwise we iterate, by choosing $i_2 = \min_{\prec}(Y \setminus Y_2)$, and so on.

Note that each generator $gen_j = \{Y_j \cup i_j\}$ is *order preserving*, because $c(\{Y_j \cup i_j\}) = Y_{j+1} \subseteq Y$ and i_j is

---

1 For each closed itemset $Y' \neq c(\emptyset)$, it is straightforward to show that there must exists at least a generator having the form $gen = Y \cup i$, where $Y, Y \subset Y'$, is a closed itemset, $i \notin Y$, and $Y' = c(gen)$.

2 This lexicographic order is induced by an order relation between single item literals, according to which each k-itemset I can be considered as a *sorted set* of k *distinct* items $\{i_0, i_1, ..., i_k\}$.

the minimum item in $\{Y \setminus Y_j\}$ by construction, i.e. $i_j \prec \{Y_{j+1} \setminus \{Y_j \cup i_j\}\}$.
□

Corollary 1 *For each closed itemset $Y \neq c(\emptyset)$, the sequence of* order preserving generators $\{gen_0, gen_1, \ldots, gen_n\} = \{Y_0 \cup i_0, Y_1 \cup i_1, \ldots, Y_n \cup i_n\}$ *as stated in Theorem 1 is unique.*

Proof. Since all the items in Y_0 appear in every transaction of the dataset, by definition of closure, they must be included also in Y, we have that $Y_0 = c(\emptyset)$.

During the construction of the sequence of generators, suppose that we choose $i_j \neq \min_{\prec}(Y \setminus Y_j)$ to construct generator gen_j. Since gen_j and all the following generators must be *order preserving*, it should be impossible to obtain Y, since we can not consider anymore the item $i = \min_{\prec}(Y \setminus Y_j) \in Y$ in any other generator or closure in order to respect the *order preserving* property.
□

Looking at Figure 1.(a), for each closed itemset we can easily identify those unique sequences of order preserving generators. For example, for the the closed itemset $Y = \{A, B, C, D\}$, we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup A$, $Y_1 = c(gen_0) = \{A, B, C\}$, $gen_1 = \{A, B, C\} \cup D$, and, finally, $Y = c(gen_1)$. Another example regards the closed itemset $Y = \{B, D\}$, where we have $Y_0 = c(\emptyset) = \emptyset$, $gen_0 = \emptyset \cup B$, $Y_1 = c(gen_0) = B$, $gen_1 = B \cup D$, and, finally, $Y = c(gen_1)$.

In order to exploit the results of Theorem 1, we need a fast way to check whether a generator is order preserving.

Lemma 3 *Let $gen = Y \cup i$ be a generator of a closed itemset where Y is a closed itemset and $i \notin Y$, and let pre-set$(gen) = \{j \prec i \mid j \notin gen\}$. gen is* not order preserving*, iff $\exists j \in$ pre-set(gen), such that $g(gen) \subseteq g(j)$.*

Proof. If $g(gen) \subseteq g(j)$, then $j \in c(gen)$. Since, by hypothesis, $j \prec i$, it is not true that $i \prec (c(gen) \setminus gen)$ because $j \in (c(gen) \setminus gen)$.
□

The previous Lemma introduces the concept of *pre-set*(gen), where $gen = \{Y \cup i\}$ is a generator, and gives a way to check the order preserving property of gen by scanning all the $g(j)$, for all $j \in$ *pre-set*(gen).

We have thus contributed a deep study on the the problem of duplicates in mining frequent closed itemsets. By reformulating the duplicates problem as the problem of visiting the lattice of frequent itemsets, according to a total (lexicographic) order, we have moved the dependencies of the order preserving check from the set of closed itemsets already mined to the *tid-lists* associated with single items. This new technique is not resource demanding, because frequent closed itemsets need not to be stored in the main memory during the computation, and it is not time demanding, because the order preserving check is cheaper than searching the set of closed itemsets mined so far. Note that CLOSET+ needs the initial FP-tree as an additional requirement the current FP-tree in use, and morover does not use its upward checking tchnique with dense datasets.

5. The *DCI_Closed* algorithm.

The pseudo-code of the recursive procedure *DCI_Closed*() is shown in Algorithm 1. The procedure receives three parameters: a closed itemsets CLOSED_SET, and two sets of items, i.e. the PRE_SET and POST_SET.

The procedure will output all the *non-duplicate closed itemsets* that properly contain CLOSED_SET. In particular, the goal of the procedure is to deeply explore each *valid new generator* obtained from CLOSED_SET by extending it with all the element in POST_SET.

Before calling procedure *DCI_Closed*(), the dataset $\mathcal{D}$ is scanned to determine the frequent single items $\mathcal{F}_1 \subseteq \mathcal{I}$, and to build the bitwise vertical dataset $\mathcal{VD}$ containing the various *tid-lists* $g(i)$, $\forall i \in \mathcal{F}_1$. The procedure is thus called by passing as arguments CLOSED_SET = $c(\emptyset)$, PRE_SET = $\emptyset$, and POST_SET = $\mathcal{F}_1 \setminus c(\emptyset)$. Note that the itemset $c(\emptyset)$ contains, if any, the items that occur in all the transactions of the dataset $\mathcal{D}$.

The procedure builds all the possible *generators*, by extending CLOSED_SET with the various items in POST_SET (lines 2–3). The *infrequent* and *duplicate* generators (i.e., the *not order preserving* ones) are however discarded as *invalid* (lines 4-5). Note that the items $i \in POST\_SET$ used to obtain those invalid generators will no longer be considered in the following recursive calls. Only the valid generators are then extended to compute their closure (lines 6–15). It is worth noting that each generator *new_gen* ← CLOSED_SET $\cup\, i$ is strictly extended according to the order preserving property, i.e. by using all items $j \in$ POST_SET such that $i \prec j$ (line 8). Note that all the items j, $i \prec j$, which do not belong to $c(new\_gen)$ are included in the new POST_SET (line 12) and are used for the next recursive call. At the end of this process, a new closed set (CLOSED_SET$_{New}$ ← $c(new\_gen)$) is obtained (line 15). From this new closed set, new generators and corresponding closed sets can be build, by recursively calling the procedure *DCI_Closed*() (line 16). Finally, it

Algorithm 1 DCI-closed pseudocode

```
1: procedure DCI_Closed(CLOSED_SET, PRE_SET, POST_SET)
2:     for all i ∈ POST_SET do                                  ▷ Try to create a new generator
3:         new_gen ← CLOSED_SET ∪ i
4:         if supp(new_gen) ≥ min_supp then                     ▷ new_gen is frequent
5:             if is_dup(new_gen, PRE_SET) = FALSE then         ▷ Duplication check
6:                 CLOSED_SET_New ← new_gen
7:                 POST_SET_New ← ∅
8:                 for all j ∈ POST_SET, i ≺ j  do              ▷ Compute closure of new_gen
9:                     if g(new_gen) ⊆ g(j) then
10:                        CLOSED_SET_New ← CLOSED_SET_New ∪ j
11:                    else
12:                        POST_SET_New ← POST_SET_New ∪ j
13:                    end if
14:                end for
15:                Write out CLOSED_SET_New and its support
16:                DCI_Closed(CLOSED_SET_New, PRE_SET, POST_SET_New)
17:                PRE_SET ← PRE_SET ∪ i
18:            end if
19:        end if
20:    end for
21: end procedure
22:
23:
24: function is_dup(new_gen, PRE_SET)
25:    for all j ∈ PRE_SET do                                   ▷ Duplicate check
26:        if g(new_gen) ⊆ g(j) then
27:            return FALSE                                     ▷ new_gen is not order preserving!!
28:        end if
29:    end for
30:    return TRUE
31: end function
```

is worth to point out that, in order to force the lexicographic order of the visit, the two **for all**'s (line 2 and line 8) have to extract items from POST_SET while respecting this order.

Before recursively calling the procedure, it is necessary to prepare the suitable PRE_SET and POST_SET to be passed to the new recursive call of the procedure. Upon each recursive call to the procedure, the size of the new POST_SET is monotonically decreased, while the new PRE_SET's size is instead increased.

As regards the composition of the new POST_SET, assume that the closed set X =CLOSED_SET$_{new}$ passed to the procedure (line 16) has been obtained by computing the closure of a generator $new\_gen = Y \cup i$ ($c(new\_gen)$), where Y =CLOSED_SET and $i \in$ POST_SET. The POST_SET$_{new}$ to be passed to the recursive call of the procedure is built as the set of all the items that follow i in the lexicographic order and that have not been already included in X. More formally, POST_SET$_{new} = \{j \in F_1 \mid i \prec j \text{ and } j \notin X\}$. This condition allows the recursive call of the procedure to only build new generators $X \cup j$, where $i \prec j$ (according to the hypotheses of Theorem 1.

The composition of the new PRE_SET depends instead on the *valid* generators[3] that precedes $new\_gen = Y \cup i$ in the lexicographic order. If all the generators were valid, it would simply be composed of all the items j that precede i in the lexicographic order, and $j \notin X = c(new\_gen)$. In other words, the new PRE_SET would be the complement set of $X \cup$ POST_SET$_{new}$.

While the composition of POST_SET guarantees that the various generators will be produced according to the lexicographic order $\prec$, the composition of PRE_SET guarantees that duplicate generators will be pruned by function *is_dup*().

Since we have shown that for each closed itemset Y exists one and only one sequence of *order preserving* generators and since our algorithm clearly explores every possible *order preserving* generator from every

3 The ones that have passed the frequency and duplicate tests.

closed itemset, we have that the algorithm is complete and does not produce any duplicate.

5.0.1. Some optimizations exploited in the algorithm. We adopted a large amount of optimizations to reduce the cost of the bitwise intersections, needed for the duplication and closure computations (line 10 and 34). For the sake of simplicity, these optimizations are not reported in the pseudo-code shown in Algorithm 1.

DCI-CLOSED inherits the internal representation of our previous works DCI[4] and kDCI[3].The dataset is stored in the main memory using a vertical bitmap representation. With two successive scans of the dataset, a bitmap matrix $D_{M\times N}$ is stored in the main memory. The $D(i,j)$ bit is set to 1 if and only if the j-th transaction contains the i-th frequent single item. Row i of the matrix thus represent the tid-list of item i.

The columns of D are then reordered to profit of data correlation. This is possible and highly worthwhile when we mine dense datasets. As in [3][4], columns are reordered to create a submatrix E of D having all its rows identical. Every operation (e.g. intersection ones) involving rows in the submatrix E will be performed only once, thus gaining strong performance improvements.

This kind of representation fits with our framework, because the three main operations, i.e. support count, closure, and duplicates check, can be fastly performed with cheap bit-wise AND/OR operation.

Besides the DCI optimizations, specifically tailored for sparse and dense datasets, we exploited more specific techniques made possible by the depth-first visit of the lattice of itemsets.

In order to determine that the itemset X is closed, the tidlist $g(X)$ must have been compared with all the $g(j)$'s, for all items j contained in the *pre-list* (*post-list*) of X, i.e. the items that precede (follows) all items included in X according to a lexicographic order. The PRE_SET must have been accessed for *checking duplicate generators*, and the POST_SET for *computing the closure*. In particular, for all $j \in$ PRE_SET $\cup$ POST_SET, we already know that $g(X) \not\subseteq g(j)$, otherwise those items j must have been included in X.

Therefore, since $g(X)$ must have already been compared with all the $g(j)$, for all items j contained in the PRE_SET (POST_SET) of X, we may save some important information regarding each comparison between $g(j)$ and $g(X)$. Such information will be used to reduce the cost of the following use of $g(j)$, when these tidlists $g(j)$ will have to be exploited to look for further closed itemsets that include/extend X. In particular, even if, for all j, it is true that $g(X) \not\subseteq g(j)$, we may know that some large portions of the bitwise tidlists $g(X)$ are however strictly included in $g(j)$. Let $\overline{g(X)}_j$ be the portion of the bitwise tidlist $g(X)$ strictly included in the corresponding portion of $g(j)$, namely $\overline{g(j)}$. Hence, since $\overline{g(X)}_j \subseteq \overline{g(j)}$, it is straightforward to show that $\overline{g(X \cup Y)}_j \subseteq \overline{g(j)}$ continues to hold, for all itemset Y used to extend X, because $g(X \cup Y) \subseteq g(X)$ holds . So, when we extend X to obtain a new generator, we can limit the inclusion check of the various $g(j)$ to the complementary portions of tid-lists $\overline{g(j)}$, thus strongly reducing the cost of them.

5.0.2. Dealing with sparse datasets. It is possible to show that in sparse datasets the number of closed itemsets is nearly equal to the number of frequent ones, so near that they are often the same. This means that the techniques for mining closed itemsets are of no use, because almost every duplicate checking or closure calculating procedure is likely to fail.

For this reason, in case of sparse datasets, we preferred to exploit our frequent itemset mining algorithm [3] with an additional closedness test over the frequent itemset discovered. Given a new frequent itemset X, every of it subset of length $|X|-1$ with the same support as X is marked as non closed. Experiments showed that this approach is fruitful (see Fig. 2.b).

5.0.3. Space complexity. For all the algorithms requiring to keep in the main memory the whole set of closed itemsets to perform the duplicate check, the size of the output is actually a lower bound on their space complexity. Conversely, we will show that the amount of memory required by an implementation based on our duplicate discarding technique is independent of the size of the output. To some extent, its memory occupation depends on those data structures that also need to be maintained in memory by other algorithms that visit depth-first the lattice and exploit tid-list intersections adopting a vertical datasets.

The main information needed to be kept in the main memory is the tid-list of each node in the current path along the lattice, and the tid-list of every frequent single item. In this way we are able to browse the search space intersecting nodes with single item tid-lists, and to discard duplicates checking the order preserving property.

The worst case of memory occupation happens when the number of generators and frequent single items is maximal: this occurs when $c(\emptyset) = \emptyset$ and every itemset is frequent and closed. If N is the number of frequent single items, the deepest path has N nodes, and since one of this node is a single item, the total number of tid-lists to be kept in the main memory is $2N-1$. Since the length of a tid-list is equal to the number of

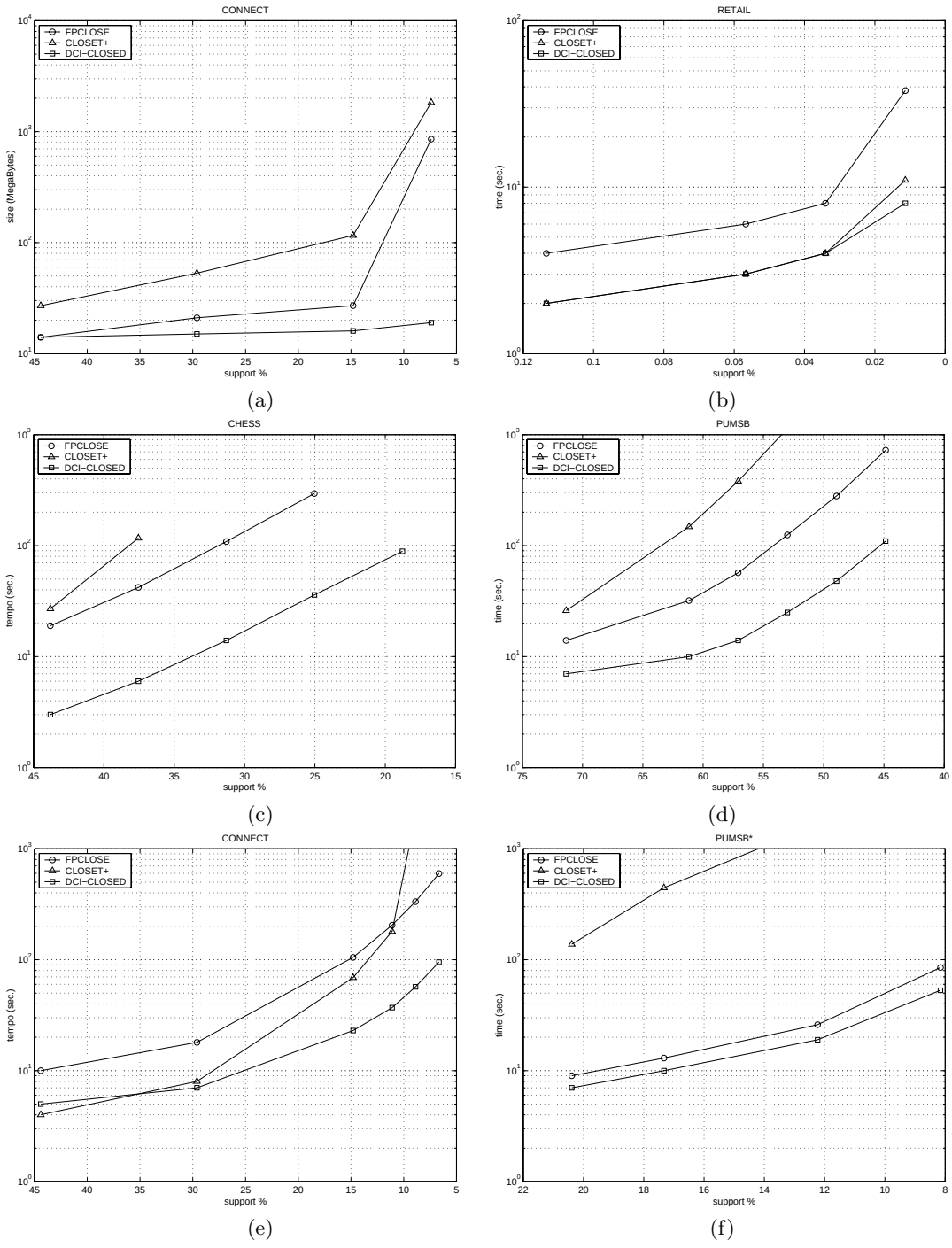


Figure 2. (a) Memory occupation on the connect dataset as a function of the minimum support threshold. (b-f) Execution times of FPCLOSE, CLOSET+, and DCI-CLOSET as a function of the minimum support threshold on various publicly available datasets.

transactions T in the dataset, the space complexity of our algorithm is

$$O\left((2N-1)\times T\right).$$

Figure 2.(a) plots memory occupation of FPCLOSE, CLOSET+ and our algorithm DCI-CLOSED when mining the `connect` dataset as a function of the support threshold. The experimental results agree with our estimates: whereas FPCLOSE and CLOSET+ memory occupation grows exponentially because of the huge number of closed itemsets generated, our implementation needs much less memory (up to two order of magnitude) because its occupation depends linearly on N.

5.1. Experimental results

We tested our implementation on a suite of publicly available dense datasets (`chess`, `connect`, `pumsb`, `pumsb*`), and compared the performances with those of two well known state of the art algorithms: FPCLOSE [1], and CLOSET+ [7]. FPCLOSE is publicly available as `http://fimi.cs.helsinki.fi/src/fimi06.html`, while the Windows binary executable of CLOSET+ was kindly provided us from the authors. Since FPCLOSE was already proved to outperform CHARM in every dataset, we did not used CHARM in our tests.

The experiments were conducted on a Windows XP PC equipped with a 2.8GHz Pentium IV and 512MB of RAM memory. The algorithms FPCLOSE and DCI-CLOSED were compiled with the gcc compiler available in the cygwin environment.

As shown in Fig. 2.(b-f), DCI-CLOSED outperforms both algorithms in all the tests conducted. CLOSET+ performs quite well on the `connect` dataset with low supports, but in any other case it is about two orders of magnitude slower. FPCLOSE is effective in `pumsb*`, where it is near to DCI-CLOSED, but it is at one order of magnitude slower in all the other tests.

6. Conclusions

In this paper we provide a deep study on the problem of mining frequent closed itemsets, formalizing a general framework fitting every mining algorithm. Use such framework we were able to analyse the problem of duplicates rising in this new mining problem.

We have proposed a technique for promptly detecting and discarding duplicates, without the need of keeping in the main memory the whole set of closed patterns, and we implemented this technique into a new algorithm which uses a vertical bitmap representation of the dataset.

The experimental evaluation demonstrated that our approach is very effective. The proposed implementation outperforms FPCLOSE and CLOSET+ in all the test conducted and requires orders of magnitude less memory.

References

[1] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[2] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD '00*, pages 1–12, 2000.

[3] Claudio Lucchese, Salvatore Orlando, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[4] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proc. The 2002 IEEE International Conference on Data Mining (ICDM02)*, page 338345, 2002.

[5] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. ICDT '99*, 1999.

[6] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD International Workshop on Data Mining and Knowledge Discovery*, May 2000.

[7] Jian Pei, Jiawei Han, and Jianyong Wang. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD '03*, August 2003.

[8] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Technical Report 01-1, Computer Science Dept., Rensselaer Polytechnic Institute*, March 2001.

[9] Mohammed J. Zaki and Ching-Jui Hsiao. Charm: An efficient algorithm for closed itemsets mining. In *2nd SIAM International Conference on Data Mining*, April 2002.